A Data Model for Querying Wavelet Features in Image Databases

Simone Santini* Amarnath Gupta†

Abstract

Multimedia databases deal with storage and retrieval of complex descriptors of image contents, called features. Traditional techniques consider features as "black boxes," often represented as vectors for which the only operation defined is distance computation. This "modus describendi" resulted in the widespread utilization of similarity queries, which rest solely on the computation of distances between feature vectors.

The capacity to express queries more complex than simple similarity, however, rests on the possibility of describing and manipulating the structure of the features. In order to achieve this, features should be described as complex data types inside the database, and opportune operators for manipulating these data types should be defined.

In this paper we try to demonstrate the efficacy and feasibility of such a program by modeling a widely used image feature: the wavelet transform. We represent features using a complex data type derived from arrays, and propose a basic algebra from which operators can be derived to manipulate and query wavelet features.

1 Introduction

Several factors distinguish multimedia database systems from relational or complex object databases. Specifically relevant to this paper is the following. Unlike a traditional database, the tasks of storage and retrieval in a multimedia database are not performed on the original "media objects", but on a set of derived objects called *features*, each produced by applying a series of transformations to the media objects. Most often, regardless of the set of transformations applied, a feature is modeled as a "vector", and the only way to use this vector is through a distance function that compares two vectors and returns a distance [8, 3, 5]. The choice of making vector comparison the sole mode of image retrieval has had mixed benefits. On the one hand, it has led to the discovery of many effective features that work remarkably well in retrieving images with specific characteristics. However, it has also has a negative effect on the data management aspect of the problem. Most of the database-oriented research has been to develop improved query languages and index structures for similarity-search, and for performing result ranking in the presence of feature-weights. A specific area that has remained underexplored is that the database almost never utilizes the internal structure of a feature. For example, while a histogram is a very popular class of feature (used in various forms such as color histograms, co-occurrence matrices, pattern spectra, image shape

*Praja, Inc. San Diego, CA, ssantini@praja.com

†San Diego Supercomputer Center, gupta@sdsc.edu

spectra etc.), a typical multimedia database would often treat a histogram as an opaque object that can only be compared to another histogram, but would usually not interpret it as an array on which patterns can be sought. Thus few image databases can address a query like: "find other images having histograms that have a sharper peak in this range of values, and a deeper valley in this other range" [4].

The ability to express queries other (and possibly more complex) than similarity queries can be achieved when the data models for image database systems can capture the semantics of the features (e.g., two histograms with different number of bins can be compared by coarsening one), and are equipped with a larger set of operators to manipulate the feature structures.

The purpose of this paper is to demonstrate the efficacy of such a data model using a popular and complex feature—the wavelet transform [1]. The model assumes that a suitable wavelet transform has already been computed on an image. It offers a method to represent the resulting feature in a manner that enhances the ability to manipulate the model and retrieve images by complex retrieval operations on this feature. This enables us not to precompute all possible features when the images are inserted, but compute a primary set of features (such as the wavelet coefficients at k levels), and compute a more targeted feature to suit the query at run time. As an example, consider a query that attempts to retrieve all images that have a texture pattern, and the feature database stores significant wavelet coefficients at multiple bands. If the texture pattern of the query image is found to be most dominant in a few of the LH wavelet bands and not in the others, it may be more efficient to select out only the qualifying LH bands to perform the distance computation rather than using a predefined feature over all bands. We illustrate however, that our model does not preclude the precomputation of any feature—in fact, it can express any similarity query based on wavelet features that have been defined in literature.

2 Preliminaries

Our data model for retrieval of wavelet features is based upon the concept of complex objects developed extensively in database literature. A complex object model assumes that a piece of data can be either an uninterpreted base type T such as integer and Boolean, or it can be an instance of an abstract data type, created by type constructor functions. Typical type constructors are *sets*, *lists*, and *tuples*, denoted respectively as $\{T\}$, $[T]$ and $\times\tau_i$, where T is any base or derived type and the tuple operation creates a "record" data type where each attribute ai may have a different data type t_i. Our data model extends an existing data model developed by Libkin, Machlin and Wong for array objects (henceforth the "LMW" model [6]) which is itself based on an extension of the Nested Relational Calculus. The LMW model considers arrays as functions from a rectangular index set to a specific data type (the *data type of the array*; arrays can be created from other arrays by specifying a function based upon a minimal set of operators. Specifically, in addition to set and tuple manipulation operators, the LMW model allows arithmetic operations on natural numbers and four operations to generate and extract an element out of natural number indexed k-dimensional arrays $[\![T]\!]_k$. The array algebra of LMW is based on three operations:

- The array constructor $[\![e | i_1 < e_1, \ldots, i_k < e_k]\!]$, where $i_h \in \mathbb{N}$, $e_h \in \mathbb{N}$ for all h, and e

is an expression generating values of type t. The constructor builds the array whose value for indices $i_1 \cdots i_k$ is $\lambda e.i_1 \cdots i_k$.

- The subscript function $e[i]$, with $i \in \mathbb{N}^k$, which computes the value of the array function for a given index.
- The dimension function $dim_k(e)$, which returns a list of all the dimensions of the array, and the derived functions $dim_{k,i}(e)$ which return the ith dimension of he array.

A fourth operation, for transforming an indexed set into an array, will not be used in this paper. The model additionally allows any Boolean-valued predicate (of the form *if* $f(x) \circ g(y)$ *then true else false*), where f, g and $\circ$ are defined on the base and constructed types.

The LMW model also admits multiple values per cell. In this paper, we will not consider this extension. However, we will allow the array to consist of another complex type. For example, if $Typecolor = red, green, blue$ is a defined data type and A is a 2D array of type color, then $A[i,j].red$ denotes the red value of the (i,j)th cell, and $A.red$ is a shorthand to construct an array only with the red component of all cells projected out. Given an array A, we will also refer to the expression that generates that array as e_A. The elements of teh arrays will then be generated by the function $e_A[i] = \lambda e_A.i$.

In this paper, we will extend this model by creating the models and operators for a new type constructor called index-constrained set on multi-dimensional arrays to model multi-band wavelet features, and by defining regions with arbitrary shapes on single and constrained array-sets. Although we will follow closely the LMW model, all the examples will be in rather standard ML [9], augmented with array operations written in a rather straightforward notation. For the sake of clarity, we will avoid the complete (and rather complex) notation of [6] which the reader is invited to consult anyway. The completion of the examples and their casting in the complete notation should be straightforward.

We would like to reiterate that the primary rationale for adopting a complex object model for multimedia features is that we believe the current approach treating features as black boxes, severely restricts the expressiveness of retrieval operations that could be applied on features, and exposing the internal structures of features through a well- defined set of types and operations alleviates this limitation. At the same time, we do not profess the use of a general-purpose programming language to perform arbitrary operations on features, because that approach reduces the applicability of efficient search structures on large sets of data.

3 Regions in Arrays

Dealing with images often requires defining regions with arbitrary shapes. For this reason, we extend the array algebra with the concepts of *region* and *shape*. In order to do this, we first introduce the "universal null value" ϕ. The value ϕ doesn't have a type, but variables of any type can take the value ϕ. Moreover, if $\circ$ is an operator $\circ : T \times U \rightarrow V$ (that is, an operator that takes values of types T and U and returns a result of type V), we assume that ϕ is identified with the neutral element of $\circ$.

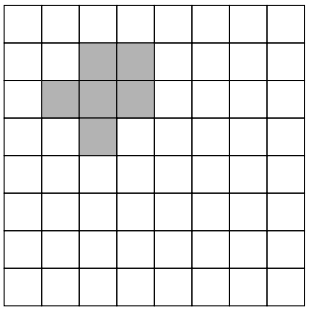

Figure 1:

Definition 3.1. *Given an expression e, a* region *S over A is a pair $S(e) = (A, E)$, where $A = [\![e | i_1 < e_1, \ldots, i_k < e_k]\!]$, and $E = \{u_1, \ldots, u_n\}$, $u_i \in \mathbb{N}^k$ is a collection of indices. The values $e_1 \cdots e_k$ are the size of the shape.*

Functionally, the region $S(e)$ is the array defined as

$$\lambda(\textit{if } i \in E \textit{ then } \lambda e.i \textit{ else } \phi).i \tag{1}$$

The region of Figure 1, for instance, is defined as $A = [\![e | 0 \leq i_1 \leq 7, 0 \leq i_2 \leq 7]\!]$,

$$E = \{(1,2), (1,3), (2,1), (2,2), (2,3), (3,2)\}$$

The dimensions of the array A are the intrinsic dimensions of the shape and, when the shape is applied to an array expression, they will be the dimensions of the resulting array. A region can also be applied to an array. In this case, the effect will be the same of applying the shape to the expression that generates the array, but the dimensions of the resulting array will be those of the array to which the shape is applied. For instance, given a two-dimensional array A of size 256×256, the expression $S(e_A)$ is a 9×9 array (the size of the shape), while the expression $S(A)$ is a 256×256 array containing the same shape and padded with ϕ values.

Formally, for a shape expression is still possible to define the function dim_k which, in this case, will return the dimension of the intrinsic size of the feature. Note that this function depends on the shape only, and is different from the same function applied to an array to which the shape function has been applied. For the shape above, for instance, it is $dim_2(S) = [9, 9]$, $dim_2(S(e_A)) = [9, 9]$, and $dim_2(A) = [256, 256]$.

Unions and intersections of regions can be defined in the standard way.

Region can betransformed into "floating" *shapes* by adding a parameter that represents their position in an array. If $j \in \mathbb{N}^k$, then a shape is a function $F(e) : \mathbb{N}^k \to [\![\tau]\!]$ that, for every instantiation of the location parameter j, generates a region with the upper-right hand corner located in position j. In other words, $F(e)(j)$ is the pair $F(e)(j) = (A, E)$ with A and E defined as before but where e generates the function

$$\lambda(\text{if } i \in E + j \text{ then } \lambda e.i - j \text{ else } \phi).i, \tag{2}$$

where $E + j = \{i + j : i \in E\}$. Note that, for correctness of notation, the shape is defined as a curried function, although in general the more common notation $F(e, j)$ will be used.

The usage of shapes outlined in this definition is analogous to the use of "masks" or "sliding masks" in image processing. Nevertheless, formally, a shape is not an array, but a mapping from array-defining expressions to arrays. A shape applied to an array-generating expression, like $s(e)$, is an array. A floating mask $F(e)$, on the other hand, is a function from indices to arrays, that is: $F(e) : \mathbb{N}^k \to [\![\tau]\!]$.

An operation often required when manipulating features is to determine whether a given property is verified for a given shape in any position of the array. This can be done using the `sweep` function, whose semantics can be defined for two dimensional arrays using monoid comprehension [2] as

$$\exists(F(e), A, P) = or\{P(F(e)(j), A) | j \leftarrow [0,0] \ldots [dim_1(A), dim_2(A)]\} \tag{3}$$

The definition can be easily extended to n dimensional arrays. Other operations can be defined in a similar way simply changing the target monoid. For instance, a selection operation, which returns all the locations in an array where the condition P is met for a shape can be defined as

$$\sigma(F(e), A, P) = set\{j | j \leftarrow [0,0] \ldots [dim_1(A), dim_2(A)], P(F(e)(j), A)\} \tag{4}$$

and the operation that counts how many times the condition P is satisfied is defined as

$$\Sigma(F(e), A, P) = +\{j | j \leftarrow [0,0] \ldots [dim_1(A), dim_2(A)], P(F(e)(j), A)\} \tag{5}$$

Shapes allows one to start expressing queries involving complex regions of the image.

Example 1. Find all arrays that agree with array A on the (localized) shape S.

```
fun within t,a,b =>
  let
    fn norm a => summapp(fn a, i => a[i]^2);
  in
    if norm a-b > t then false else true
query close_in_region a, s =>
  { d | d in ArrayDb.array1 and within t s(d) s(a) }
```

In this query, "ArrayDb" is the database, and "ArrayDb.array1" isolates a column that has been declared of array type.

Example 2. Find all arrays of size (256,256) where shape pattern $S(e_A)$ (A being a predefined array) occurs at least twice within the region (128,128), (255,255).

```
query shape_occur S t =>
  let
    fun P d t F =>
      within (t, F, d);
  in
  { d |
    dim(d, 1) = 256 and dim(d, 2) = 256 and
    count(128, 128, d, S, P(d, t)) >= 2 }
```

4 From Arrays to Index Constrained Arrays

Structurally, many wavelet transforms can be described as sets of arrays in which certain relations exists between elements of different arrays, namely, there are elements of different arrays that represent the same "physical location" in the image.

In order to arrive to such definition, we begin with the concept of *index constrained arrays*.

Definition 4.1. *An array B index-constrains an array B through the function f, written $A \overset{f}{\Longrightarrow} B$ if $A : [\![T]\!]_k$, $B : [\![M]\!]_k$, and $f : \mathbb{N}^k \to \mathbb{N}^k$ are such that*

1. *For all i, $dim_{k,i}(A) \leq dim_{k,i}(B)$*
2. *The function f is defined for all legal values of indices of A, and for all such indices i, $f(i)$ is a legal index for B.*

A pair of arrays with the same size along all the dimensions are called iso-dimensional. The trivial index constrained pair for two iso-dimensional arrays A and B is $A \overset{\iota}{\Longrightarrow} B$, where ι is the identity transformation $\iota : \mathbb{N}^k \to \mathbb{N}^k$.

The inverse map f^{-1} maps elements of A into subsets of elements of B. The set of elements $f^{-1}(i)$, $i \in \mathbb{N}^k$ is called the *constraining set* for the element $A[i]$. Similarly, given a set of indices of A, $Q = \{i_1, \ldots, i_m\}$, the set $f^{-1}(Q)$ is the constraining set of the set $\{A[i_1], \ldots, A[i_m]\}$.

Note that the presence of the function f and its invertibility (in the set-of-indices sense illustrated aove) places some constraints on the sizes of the arrays. For instance, if A and B are two-dimensional arrays and $A \overset{f}{\Longrightarrow} B$ is an index constrained pair with $f(i,j) = (\lfloor i/2 \rfloor, \lfloor j/2 \rfloor)$, then if A has size $n \times n$, B must have size $2n \times 2n$. In this sense, the presence of the function f represents a structural constraint between pairs of arrays.

The definition of index constraint pair can be extended to sets of functions. Consider a set $\mathfrak{F}$ of functions $\mathbb{N}^k \to \mathbb{N}^k$ with the structure of a semi-group, that is $\iota \in \mathfrak{F}$ and $f_1, f_2 \in \mathfrak{F} \Leftarrow f_1 \circ f_2 \in \mathfrak{F}$. A pair of arrays is index constrained by the set $\mathfrak{F}$, written $A \overset{\mathfrak{F}}{\Longrightarrow} B$ if they are index constrained by any of the functions in $\mathfrak{F}$, that is,

$$A \overset{\mathfrak{F}}{\Longrightarrow} B \Rightarrow \exists f \in \mathfrak{F} : A \overset{f}{\Longrightarrow} B \tag{6}$$

Constrainment by a semi-group $\mathfrak{F}$ is reflexive and transitive. It is reflexive since for every array A it is trivially $A \overset{\iota}{\Longrightarrow} A$ (and, since $\mathfrak{F}$ is a semi-group, $\iota \in \mathfrak{F}$), and reflexive since $A \overset{f_1}{\Longrightarrow} B$ and $B \overset{f_2}{\Longrightarrow} C$ implies $A \overset{f_2 \circ f_1}{\Longrightarrow} C$.

Consider now a set of arrays $P = \{A^1, \ldots, A^N\}$ such that there are pairs of arrays A^u, A^v in P such that $A^u \overset{f}{\Longrightarrow} A^v$ for a function f belonging to a predefined semigroup $\mathfrak{F}$. Such a set is called an *index constrained set of arrays*, and will be denoted as $\langle P, \mathfrak{F} \rangle$.

The set of functions $\mathfrak{F}$ induces a structure in the set P. An array $A \in P$ is *initial* if there is no array $B \in P$ such that $B \overset{f}{\Longrightarrow} A$ for some $f \in \mathfrak{F}$. An array $A \in P$ is *terminal* if there is no array $B \in P$ such that $A \overset{f}{\Longrightarrow} B$ for some $f \in \mathfrak{F}$.

Definition 4.2. *An index-constrained set of arrays P is* linear *if*

1. *There is one and only one terminal array* $A \in P$,

2. *For any array* $A \in P$, *if* $A \stackrel{f}{\Longrightarrow} A$ *then* f *is the identity.*

The second point in the definition implies that, with the exception of the trivial correspondence between iso-dimensional arrays, there are no "loops" in P that is, the structure induced in P by the set $\mathfrak{F}$ (again, discarding the trivial constraints) is a tree.

Given an index-constrained set $C = \langle P, \mathfrak{F} \rangle$, two arrays A and B, and and index j for A, the function $\mathrm{map}(A, B, j)$ returns the set of indices in the array B that are constrained to the index j of A. In other words:

$$\begin{aligned} \mathrm{map}(A, B, f, j) \quad = \quad & set\,\{i | j \leftarrow [0,0], \ldots, [dim_1(A), dim_2(A)], \\ & i \leftarrow [0,0], \ldots, [dim_1(B), dim_2(B)], \\ & j = f(i) \vee i = f(j)\} \end{aligned} \tag{7}$$

Similarly, the function vmap returns an array built with the values indexed by the indices returned by map:

$$\mathrm{vmap}(A, B, f, j) = array\{B[i] | i \in \mathrm{map}(A, B, f, j)\} \tag{8}$$

The definition can be generalized so that, instead of a single index j one consider a whole region in the array A: the function $map(C, B, f, R(A))$, $f \in \mathfrak{F}$ returns the region on B induced by the function f applied to the shape $S(A)$:

$$\mathrm{map}(C, B, f, R(A)) = \bigcup_{j \in R(A)} map(A, B, f, j) \tag{9}$$

Given a mapping f from A to B, in order to determine the ith element of the array $S'(B)$, one looks at all the elements of $S(A)$ that map to that element: if at least half of them are not null, then the value is $B[i]$, otherwise it is ϕ. If it is not the case that $A \stackrel{f}{\Longrightarrow} B$, then $S'(B)$ is the null array. Similarly, it is possible define the function $map(C, f^{-1}, B, S(A))$

In addition to the mapping function, several utility functions are defined. Given a list of arrays, the function `toList` transforms it into a list of sets of iso-dimensional arrays in such a way that the largest arrays come in first position, the second largest in second position, and so on. The function `first` extracts the set of the largest arrays, while the function `rest` extracts the rest of the list.

Also, given a set of arrays, the function `maxSize` extracts the subset of arrays with maximal size, and the function `minSize` extracts the set of arrays with minimal size.

As an example of applications, we will consider the definition of a multiresolution shape. Consider first a function `flt` that applies a shape expression to all arrays of a set:

```
fun flt( P, S) =
  if |P| = 0
    null
  else if |P| = 1
    S(P)
```

```
  else
    let A = randomElement(P);
    in
      S(A) ∪ flt(P − {A})
```

The function `splice` is then defined as:

```
fun splice(P, f, S) =
  if P = []
    []
  else
  flt(first(P, S)) ++ splice(rest(P), f, S ∘ f);
```

5 Wavelet Features

We assume the readers to have a basic knowledge of wavelet functions and their properties [1]. Specifically, we develop on the idea that a wavelet function transforms an original signal into multiple frequency bands, including a low-pass band. Often, in an iterative process, the low-pass band is transformed again into similar frequency bands. Sometimes, the range of the signal of the ith iteration is reduced by half at the $(i+1)$th iteration. In this section we describe how wavelet features computed on multidimensional signals is modeled using the framework of index- constrained arrays developed in the previous section.

Let us define a labeled array as a tuple $\{\lambda, A\}$ such that the label λ is assigned from a domain Λ and A is an array. Let A_0 be the base array representing the original signal for which the wavelet transform is computed. A wavelet transform $W(A_0)$ is the linear index-constrained set of iso-dimensional sets of labeled arrays denoted as

$$W(A_0) = \langle \left[A_0, \{(\lambda_1^0, A_1^0), (\lambda_1^1, A_1^1), \ldots\}, \{(\lambda_2^0, A_2^0), (\lambda_2^1, A_2^1), \ldots\} \ldots\right], f\rangle \tag{10}$$

Each linear index-constrained set $\{(\lambda_j^0, A_j^0), (\lambda_j^1, A_j^1), \ldots\}$ is an iso-dimensional set that represents the frequency bands computed at any given resolution. The label of an array serves to identify the position of the array within the resolution level. If W is a Haar wavelet, then the label belongs to the domain $\{LH, HL, HH, LL\}$, on the other hand if W is a Gabor wavelet then its domain is a discrete set of angles ranging between 0 and 360. The index mapping function f is usually a transform that reduces an array of the previous resolution by half in each of its k dimensions, that is, for two dimensional arrays $f(i, j) = (\lfloor i/2, j/2 \rfloor)$. Additional constraints may be defined on the set depending on the nature of the wavelet transform. For example, for Gabor wavelets, an array at the i-th level can constrain the index of an array at the $(i+1)$-th level only if their labels are the same.

The operations defined on wavelet features are the same as those defined on index-constrained array, with the addition of the label selection operator σ_P^λ which, given a wavelet returns an index-constrained array containing all the elements of the wavelet that satisfy the condition P.

The previously defined operations on index-constrained set of array, in addition with the usual set, list, and selection operations, allow us to express complex queries involving wavelets.

Example 1. Find all images that agree with image I (represented as a wavelet transform) in the region S at a coarse resolution.

```
fun b W, l, i =>
  if i = l
    W
  else
    b(rest(W), l, i + 1)
fun isolate S, l, W =>
  splice(b(W, l, 0), W.f, S);

query close_in_region W, l, s, t =>
  { I | I in ImageDb.wavelet and
    within (t, isolate(S, l, W), isolate(S, l, I) }
```

The function `b` takes the coarser resolution bands of the wavelet, while the function `isolate` projects the region S onto these bands. The functions `splice` and `within` are defined in the previous section.

Example 2. Find an image that agrees with image W in the shape $S1(i_0, j_0)$ and in the shape $S2(i_0, j_0)$, independently of the mutual positioning of the shapes. (This region based query is an extension of similar queries defined in [7]; the query can be expanded by defining functions that compare the relative positions of pairs of regions so that it can be imposed that the two regions be in the same spatial relation.)

```
fun cmp1 (W, I, F, i, j, t) =>
  let
    fun P t, W, I => within(t, W, I)
  in
  sweep(0, 0, I, S, P(S(W)(i, j)));
query two_regions W, F0, i0, j0, F1, i1, j1 =>
  { I | I in ImageDb.wavelet and
    cmp1 (W, I, F0, i0, j0) and cmp1 (W, I, F1, i1, j1) }
```

6 Conclusions

In this paper we have proposed a data model for the representation, manipulation and query of wavelet features in multimedia databases. We argued that the representation of features as complex data types and the possibility to manipulate their structure is important for the design of powerful and efficient multimedia databases capable of transcending the limitations of the query-by-example model.

The model we presented is based on an extension of the array data type to include sets of arrays with certain structural relations. WE have showed that it is possible define wavelets as data types within such a framework and that a minimal set of operations (namely the array operations augmented with the `map` function) can be used to express complex queries involving wavelets.

The formalization of the structure of features and the definition of an algebra for their manipulation has two orders of advantages. First, the algebra is based on a limited number of operations on sets of labelled index-constrained arrays. The index-constrained arrays can be indexed using techniques that allow an efficient computation of these operation for large amounts of data, therefore making the computation of the individual operations efficient. Preliminary work carried out by us has indicated that efficient implementation of array selection operations can be obtained both by specialized indices and by special array representation in a relational database.

Second, by exposing the internal structure and the semantics of the operations, we create numerous possibilities for query optimization. Techniques for program normalization and query optimization based on the monoid comprehension calculus have been studied for quite some time [2], and our wavelet feature definition fits nicely in this framework.

References

[1] Ingrid Daubechies. *Ten lectures on wavelets*. Society for Industrial and Applied Mathematics, Philadelphia, 1992.

[2] Leonidas Fegaras and David Maier. Towards an effective calculus for object query languages. In *Proceedings of SIGMOD '95, San Jose, CA*, pages 47–58, 1995.

[3] Myron Flickner, Harpreet Sawhney, Wayne Niblack, Jonathan Ashley, Qian Huang, Byron Dom, Monika Gorkani, Jim Hafner, Denis Lee, Dragutin Petkovic, David Steele, and Peter Yanker. Query by image and video content: the QBIC system. *IEEE Computer*, 1995.

[4] Amarnath Gupta and Simone Santini. Towards feature algebras in visual databases: The case for a histogram algebra. In *Proceedings of the IFIP Working Conference on Visual Databases (VDB5), Fukuoka (Japan)*, May 2000.

[5] Charles E. Jacobs, Adam Finkelstein, and Savid H. Salesin. Fast multiresolution image querying. In *Proceedings of SIGGRAPH 95, Los Angeles, CA*. ACM SIGGRAPH, New York, 1995.

[6] Leonid Libkin, Rona Machlin, and Limsoon Wong. A query language for multidimensional arrays: design, implementation and optimization techniques. In *Proceedings of SIGMOD '96, Montreal, Canada*, pages 228–239, 1996.

[7] Apostol Natsev, Rajeev Rastogu, and Kyuseok Shim. WARLUS: A similarity retrieval algorithm for image databases. In *Proceedings of SIGMOD '99, Philadelphia, PA*, pages 395–406, 1999.

[8] Arnold Smeulders, Marcel Worring, Simone Santini, Amarnath Gupta, and Ramesh Jain. Image databases at the end of the early years. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(1), 2001.

[9] Jeffrey Ullman. *Elements of ML Programming*. Prentice Hall, 1994.